User's Guide

Version 1.0

ParaSoft Corporation
2031 S. Myrtle Ave.
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

PARASOFT END USER LICENSE AGREEMENT

REDISTRIBUTION NOT PERMITTED

This Agreement has 3 parts. Part I applies if you have not purchased a license to the accompanying software (the "SOFTWARE"). Part II applies if you have purchased a license to the SOFTWARE. Part III applies to all license grants. If you initially acquired a copy of the SOFTWARE without purchasing a license and you wish to purchase a license, contact ParaSoft Corporation ("PARASOFT"):

(626) 305-0041

(888) 305-0041 (USA only)

(626) 305-9048 (Fax)

info@parasoft.com

http://www.parasoft.com

PART I -- TERMS APPLICABLE WHEN LICENSE FEES NOT (YET) PAID GRANT.

DISCLAIMER OF WARRANTY.

Free of charge SOFTWARE is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties of merchantability, fitness for a particular purpose and non-infringement. The entire risk as to the quality and performance of the SOFTWARE is borne by you. Should the SOFTWARE prove defective, you and not PARASOFT assume the entire cost of any service and repair. This disclaimer of warranty constitutes an essential part of the agreement. SOME JURISDICTIONS DO NOT ALLOW EXCLUSIONS OF AN IMPLIED WARRANTY, SO THIS DISCLAIMER MAY NOT APPLY TO YOU AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

PART II -- TERMS APPLICABLE WHEN LICENSE FEES PAID

GRANT OF LICENSE.

PARASOFT hereby grants you, and you accept, a limited license to use the enclosed electronic media, user manuals, and any related materials (collectively called the SOFTWARE in this AGREEMENT). You may install the SOFTWARE in only one location on a single disk or in one location on the temporary or permanent replacement of this disk. If you wish to install the SOFTWARE in multiple locations, you must either license an additional copy of the SOFTWARE from PARASOFT or request a multi-user license from PARASOFT. You may not transfer or sub-license, either temporarily or permanently, your right to use the SOFTWARE under this AGREEMENT without the prior written consent of PARASOFT.

LIMITED WARRANTY.

PARASOFT warrants for a period of thirty (30) days from the date of purchase, that under normal use, the material of the electronic media will not prove defective. If, during the thirty (30) day period, the software media shall prove defective, you may return them to PARASOFT for a replacement without charge.

THIS IS A LIMITED WARRANTY AND IT IS THE ONLY WARRANTYMADE BY PARASOFT. PARASOFT MAKES NO OTHER EXPRESS WARRANTY AND NO WARRANTY OF NONINFRINGEMENT OF THIRD PARTIES' RIGHTS. THE DURATION OF IMPLIED WARRANTIES, INCLUDING WITHOUT LIMITATION, WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE, IS LIMITED TO THE ABOVE LIMITED WARRANTY PERIOD; SOME JURISDICTIONS DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO LIMITATIONS MAY NOT APPLY TO YOU. NO PARASOFT DEALER, AGENT, OR EMPLOYEE IS AUTHORIZED TO MAKE ANY MODIFICATIONS, EXTENSIONS, OR ADDITIONS TO THIS WARRANTY.

If any modifications are made to the SOFTWARE by you during the warranty period; if the media is subjected to accident, abuse, or improper use; or if you violate the terms of this Agreement, then this warranty shall immediately be terminated. This warranty shall not apply if the SOFTWARE is used on or in conjunction with hardware or software other than the unmodified version of hardware and software with which the SOFTWARE was designed to be used as described in the Documentation. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY HAVE OTHER LEGAL RIGHTS THAT VARY BY JURISDICTION.

YOUR ORIGINAL ELECTRONIC MEDIA/ARCHIVAL COPIES.

The electronic media enclosed contain an original PARASOFT label. Use the original electronic media to make "back-up" or "archival" copies for the purpose of running the SOFTWARE program. You should not use the original electronic media in your terminal except to create the archival copy. After recording the archival copies, place the original electronic media in a safe place. Other than these archival copies, you agree that no other copies of the SOFTWARE will be made.

TERM.

This AGREEMENT is effective from the day you install the SOFTWARE and continues until you return the original SOFTWARE to PARASOFT, in which case you must also certify in writing that you have destroyed any archival copies you may have recorded on any memory system or magnetic, electronic, or optical media and likewise any copies of the written materials.

CUSTOMER REGISTRATION.

PARASOFT may from time to time revise or update the SOFTWARE. These revisions will be made generally available at PARASOFT's discretion. Revisions or

notification of revisions can only be provided to you if you have registered with a PARASOFT representative or on the ParaSoft Web site. PARASOFT's customer services are available only to registered users.

PART III -- TERMS APPLICABLE TO ALL LICENSE GRANTS

SCOPE OF GRANT.

DERIVED PRODUCTS.

Products developed from the use of the SOFTWARE remain your property. No royalty fees or runtime licenses are required on said products.

PARASOFT'S RIGHTS.

You acknowledge that the SOFTWARE is the sole and exclusive property of PARASOFT. By accepting this agreement you do not become the owner of the SOFTWARE, but you do have the right to use the SOFTWARE in accordance with this AGREEMENT. You agree to use your best efforts and all reasonable steps to protect the SOFTWARE from use, reproduction, or distribution, except as authorized by this AGREEMENT. You agree not to disassemble, de-compile or otherwise reverse engineer the SOFTWARE.

SUITABILITY.

PARASOFT has worked hard to make this a quality product, however PARASOFT makes no warranties as to the suitability, accuracy, or operational characteristics of this SOFTWARE. The SOFTWARE is sold on an "as-is" basis.

EXCLUSIONS.

PARASOFT shall have no obligation to support SOFTWARE that is not the then current release.

TERMINATION OF AGREEMENT.

If any of the terms and conditions of this AGREEMENT are broken, this AGREEMENT will terminate automatically. Upon termination, you must return the software to PARASOFT or destroy all copies of the SOFTWARE and Documentation. At that time you must also certify, in writing, that you have not retained any copies of the SOFTWARE.

LIMITATION OF LIABILITY.

You agree that PARASOFT's liability for any damages to you or to any other party shall not exceed the license fee paid for the SOFTWARE.

PARASOFT WILL NOT BE RESPONSIBLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OF THE SOFTWARE ARISING OUT OF ANY BREACH OF THE WARRANTY, EVEN IF PARASOFT HAS BEEN ADVISED OF SUCH DAMAGES. THIS PRODUCT IS SOLD "AS-IS".

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU. YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

ENTIRE AGREEMENT.

This Agreement represents the complete agreement concerning this license and may be amended only by a writing executed by both parties. THE ACCEPTANCE OF ANY PURCHASE ORDER PLACED BY YOU IS EXPRESSLY MADE CONDITIONAL ON YOUR ASSENT TO THE TERMS SET FORTH HEREIN, AND NOT THOSE IN YOUR PURCHASE ORDER. If any provision of this Agreement is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. This Agreement shall be governed by California law (except for conflict of law provisions).

All brand and product names are trademarks or registered trademarks of their respective holders.

Copyright 1993-2001

ParaSoft Corporation

2031 South Myrtle Avenue

Monrovia, CA 91016

Printed in the U.S.A, April 17, 2001

Jcontract User's Guide Table of Contents

Getting Started

Welcome 1
Installation 3
Contacting ParaSoft 4
Using Jcontract 6
Using Jcontract: A Simple Example 10
Jcontract's Monitors 17
The Log File 21
About Design by Contract 23
The Design by Contract Specification Language 26

Customizing Jcontract

Jcontract Preferences 37
Runtime Handlers 43

Man Pages

dbc_javac 47
dbc_preferences 50

Index

Index 53

Welcome

Welcome to Jcontract, a tool that enables Design by Contract (DbC) in Java.

Jcontract instruments and compiles DbC-commented Java code, then automatically checks whether contracts specified in the DbC comments are violated at runtime. Jcontract is independent of Jtest, but the two tools are complementary. After you have used Jtest to verify that the class or component is solid and correct at the unit-level, you can use Jcontract to verify system-level functionality and check whether the system misuses specific classes or components.

Jcontract uses a unique `dbc_javac` compiler that is a Design by Contract-enabled replacement for `javac`; it checks the DbC specification in the Javadoc comments, generates instrumented .java files with extra code to check the contracts in the Javadoc comments, and compiles the instrumented .java files with the `javac` compiler. This process is completely transparent; the only difference between using `javac` and `dbc_javac` is that with `dbc_javac`, the resulting .class files are instrumented with extra bytecodes to check the contracts at runtime.

After files are compiled and instrumented, Jcontract checks the contracts at runtime, and reports any violations found and stack trace information in the Jcontract GUI Monitor, the Jcontract TEXT Monitor, or a file. This helps you determine exactly when and where a violation occurs.

Jcontract's degree of program interference is completely customizable. By default, Jcontract uses a completely non-intrusive Runtime Handler that reports violations found, but does not alter program execution. You can also choose a Runtime Handler that throws exceptions when violations occur, choose a Runtime Handler that logs violations in a file, or create a customized Runtime Handler that is specially tailored to your unique needs.

Jcontract also adapts to your needs by letting you select which contract conditions you want it to instrument. This way you can optimize application performance by having Jcontract only monitor the exact conditions that are relevant at your current stage of the development process. For example, after a well-tested class is integrated into the application, you

might only want to instrument and check preconditions that verify whether the application uses the class correctly.

Installation

Installing Jcontract

To install Jcontract on a Windows machine:

1. Run the setup executable that you downloaded from the ParaSoft Web site or that is on your CD.
2. Follow the installation program's onscreen directions. The installation program will automatically install Jcontract on your system.

You must install a license before you start using Jcontract.

Installing a License

To install a license:

1. Call **1-888-305-0041** to receive your license.
2. Open the License window by choosing **Start> Programs> Jcontract 1.0> License**.
3. In the License window, enter the expiration date and password.
4. Click **Set** to set and save your license.

Contacting ParaSoft

ParaSoft is committed to providing you with the best possible product support for Jcontract. If you have any trouble installing or using Jcontract, please follow the procedure below in contacting our Quality Consulting department.

- Check the manual.
- Be prepared to recreate your problem.
- Know your Jcontract version.

Contact Information

- **USA Headquarters**

  Tel: (888) 305-0041

  Fax: (626) 305-9048

  Email: jcontract@parasoft.com

  Web Site: http://www.parasoft.com

- **ParaSoft France**

  Tel: +33 (0) 1 64 89 26 00

  Fax: +33 (0) 1 64 89 26 10

  Email: jtest@parasoft-fr.com

- **ParaSoft Germany**

  Tel: +49 (0) 78 05 95 69 60

  Fax: +49 (0) 78 05 95 69 19

  Email: quality@parasoft-de.com

- **ParaSoft UK**

  Tel: +44 171 288 66 00

Fax: +44 171 288 66 02
Email: quality@parasoft-uk.com

Using Jcontract

You can use Jcontract to instrument and compile any .java file with or without Design by Contract comments.

Using Jcontract involves two main steps:

1. Compiling your code with the special Jcontract compiler.
2. Running the program in the normal manner.

These steps are described below in “Running Jcontract on Files With DbC Contracts” on page 8 and “Running Jcontract on Files Without DbC Contracts” on page 9.

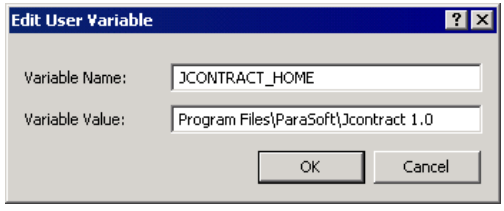
Before you use Jcontract for the first time, you need to set the environment for Jcontract.

Important: Jcontract requires JDK 1.2 or higher (JDK 1.3 is preferred).

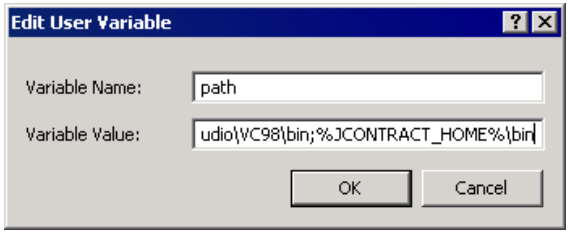
Setting the Environment

Before you use Jcontract, you need to set your environment as follows:

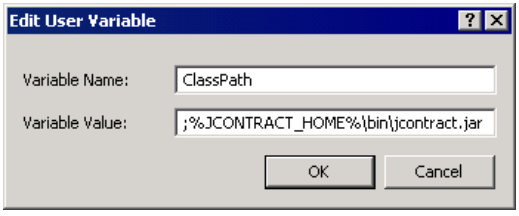
1. Set the environment variable "JCONTRACT_HOME" to the directory where Jcontract was installed.



2. Prepend "%JCONTRACT_HOME%\bin" to the "Path" environment variable.



3. Prepend "%JCONTRACT_HOME%\bin\jcontract.jar" to the "CLASSPATH" environment variable.



You can also set the above environment by opening a DOS window, changing directories to the Jcontract installation directory, then entering the following command at the prompt:

```
jcvars
```

Or, if you are using a UNIX-like shell, enter the following command:

```
. jcvars.sh
```

Adding Contracts to Your Code

Jcontract can be used to perform some checks on code without DbC contracts, but to get the full benefit of Jcontract, you should add DbC contracts to your code.

For a general introduction to Design by Contract, see “About Design by Contract” on page 23. For information on adding Design by Contract comments to your code, see “The Design by Contract Specification Language” on page 26.

Running Jcontract on Files With DbC Contracts

If you use Jcontract with code that contains DbC comments, it will instrument the comments and check the contracts.

To use Jcontract on code that contains contracts:

1. Compile the program classes using the `dbc_javac` command instead of the `javac` command.

   The `dbc_javac` compiler will then instrument the DbC comments as it compiles the program’s classes. When this process is completed, Jcontract will report the number of files compiled and the number of files instrumented.

   For example, to compile and instrument the DbC comments in Example.java, you could enter the following command at the prompt:
   ```
   dbc_javac Example.java
   ```

   For more information about the `dbc_javac` command, see “dbc_javac” on page 47.

2. Run the program in the normal manner.

   The Jcontract runtime will check the contracts and report progress and contract violations in the Jcontract Monitor (or in any other output location you have selected).

   For example, to run Example.class, you could enter the following command at the prompt:
   ```
   java Example
   ```

Running Jcontract on Files Without DbC Contracts

If you use Jcontract on code without DbC comments (i.e. code that doesn't yet contain @pre/@post/@invariant/@concurrency, etc. tags), you can have it check whether any methods are executed concurrently by more than one thread.

To use Jcontract on code that does not contain contracts:

1. Compile the program classes by entering the following command at the prompt:
   `dbc_javac -Zdefault_concurrency sequential ...`

   This will add code to check that no methods are executed concurrently by more than one thread.
2. Run the program in the regular manner.
   If more that one thread is executing a method at a given time, a contract violation will be reported in the Jcontract Monitor (or in any other output location you have selected).

Using Jcontract: A Simple Example

Introduction

The following example demonstrates how to use Jcontract to instrument and compile a simple class, then check its contracts at runtime.

Before you try this example on your own system, make sure that you have already set your environment variables for Jcontract (as described in “Setting the Environment” on page 6).

A Simple Example

The Example.java File

In this example, we check the contract of the following simple example file, Example.java:

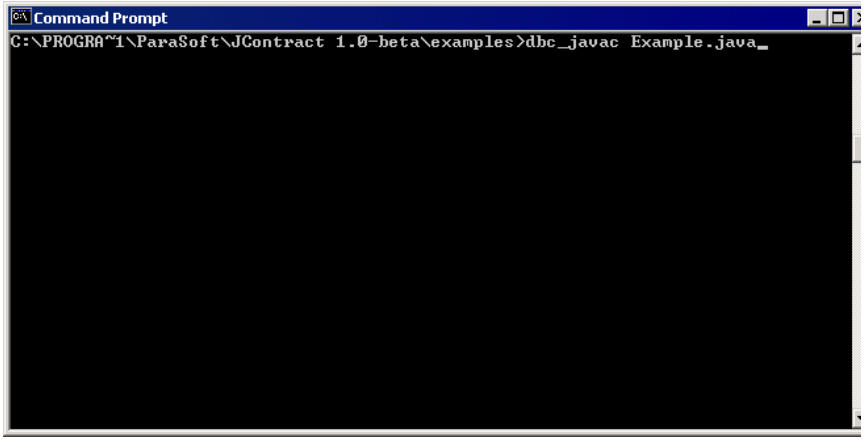
```
public class Example
{
    /** @pre month >= 1 && month <= 12 */
    static void setMonth (int month) {
        // ...
    }


    //////////

    public static void main (String[] args)
    {
         setMonth (13);
    }
}
```

If you look at the contract in this code, you'll see that it contains a @pre tag that states that the month value must be an integer between 1 and 12. Preconditions check whether or not a method is called correctly. They are checked right before the method is called, and they can access anything accessible from the method scope except for local variables.

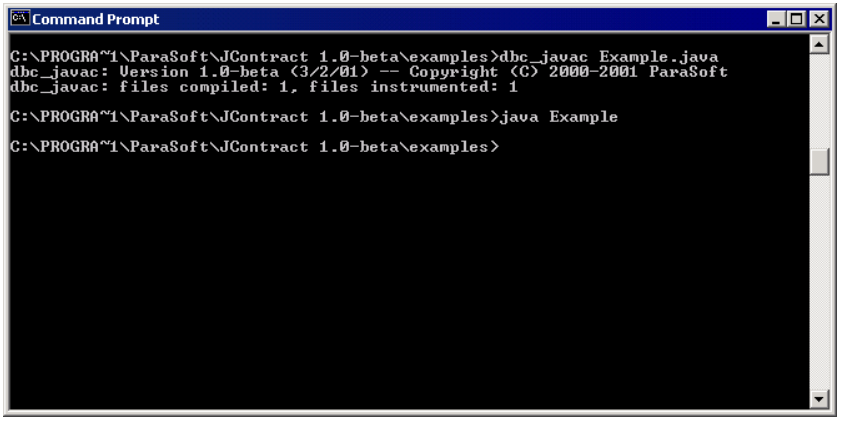
Checking the Contract

To check if this contract is met, compile the class with Jcontract's `dbc_javac` compiler, then run it as normal. To do this, perform the following steps:

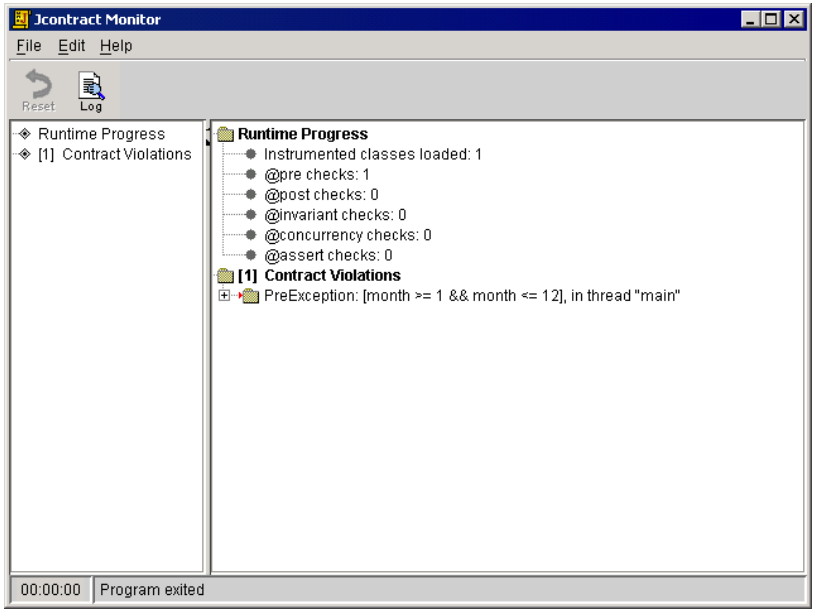
1. Compile the class by entering the following command at the prompt:
   `dbc_javac Example.java`



2. Run the class by entering the following command at the prompt:
   `java Example`


Command Prompt
C:\PROGRA~1\ParaSoft\JContract 1.0-beta\examples>dbc_javac Example.java
dbc_javac: Version 1.0-beta (3/2/01) -- Copyright (C) 2000-2001 ParaSoft
dbc_javac: files compiled: 1, files instrumented: 1
C:\PROGRA~1\ParaSoft\JContract 1.0-beta\examples>java Example
C:\PROGRA~1\ParaSoft\JContract 1.0-beta\examples>

If you have not modified the Jcontract default setting, the Jcontract GUI Monitor will then open.

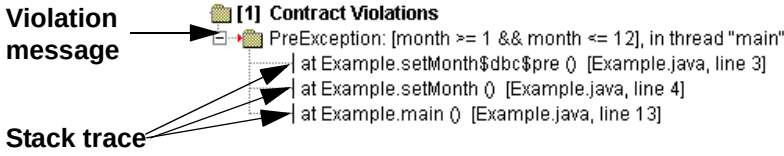


Exploring Results

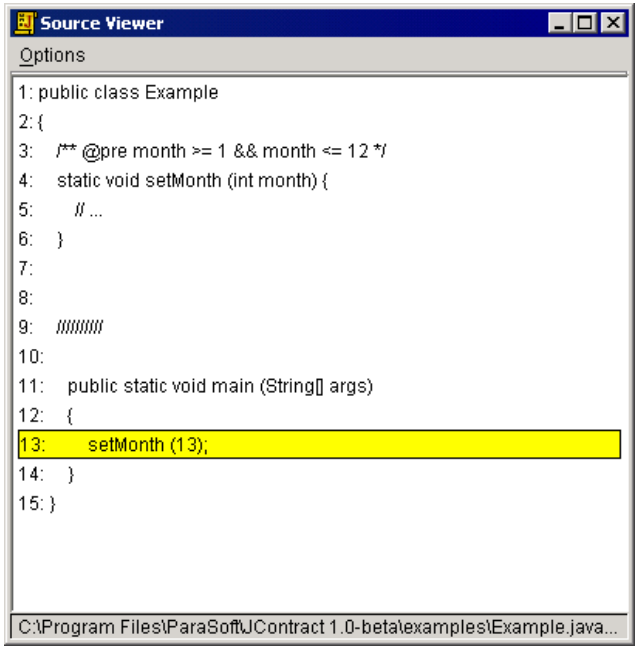
By default, the Jcontract Monitor reports runtime progress and lists any contract violations found. In addition, all result information is saved in a Jcontract log file. For more information about this log file, see “The Log File” on page 21.

If you look at the Jcontract Monitor, you will see that Jcontract found one contract violation in this example. This violation is listed in the **Contract Violations** branch in the right side of the monitor. The violation message reveals that the @pre contract condition that stated that the month value must be between 1 and 12 was not met and that the method was called incorrectly.

To learn more about the violation found, expand that branch by clicking the plus sign to the left of the violation message.


Violation message
Stack trace
[1] Contract Violations
PreException: [month >= 1 && month <= 12], in thread "main"
at Example.setMonthdbcpre () [Example.java, line 3]
at Example.setMonth () [Example.java, line 4]
at Example.main () [Example.java, line 13]

The stack trace information reveals that the violating value was set in line 13. To see the code in a source viewer, with line 13 highlighted, double-click the stack trace line that refers to line 13 of Example.java



This line of code sets the month value to 13. Because the @pre condition stated that the month value had to be an integer between 1 and 12, this value violates the contract.

Fixing Errors Found

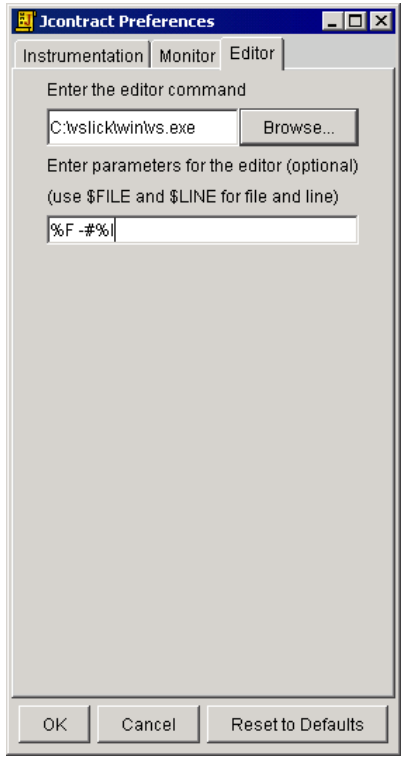
When you are ready to fix any Design by Contract violation found, there are two main things you'll want to do:

1. Examine the code to determine if there is a problem with the code or a problem with the contract.
2. If the problem was with the code, fix the code; if the problem was with the contract, modify the contract.

In this case, the problem appears to be with the code. If you wanted to modify this code, you could do so by right-clicking any line of the stack trace information, then choosing **Edit Source** from the shortcut menu. This opens the code in the default editor (WordPad).

Note: You can configure Jcontract to open code in your preferred Source Editor by performing the following steps:

1. Open the JContract preferences tab in one of the following ways:
   - In the Jcontract Monitor, choose **Edit> Preferences**.
   - Enter the following command at the command prompt: `dbc_preferences`
   - Choose **Start> Programs> Jcontract> Edit Preferences**.
2. In the Editor tab, enter the command for your preferred editor as well as any parameters you want to pass to that editor.



3. Click **OK**.

Additional Examples

A number of example .java files that contain DbC contracts are available in <jcontract_install_dir>/examples. To learn more about Jcontract, try compiling and running these examples on your system.

Jcontract's Monitors

Jcontract's monitors report contract violations detected at runtime and program progress (i.e. the number of contract checked).

Currently there are two types of monitors available with Jcontract: a TEXT mode monitor and a GUI mode monitor.

The GUI Monitor is used by default. You can specify which type of monitor you would like to use by modifying the Jcontract Preferences. See "Jcontract Preferences" on page 37 for information about how to set monitor preferences.

GUI Monitor

If the "Monitor.Type" preference in `<jcontract_install_dir>\u\<username>jcontract.preferences` is set to the value "GUI", the Jcontract GUI Monitor will start as soon as the program under execution loads a class with instrumented contracts on it.

The GUI Monitor consists of the following elements:

- The Report Area
- The Menu Bar
- The Tool Bar
- The Status Bar

The Report Area

The Report Area of the Jcontract GUIMonitor displays:

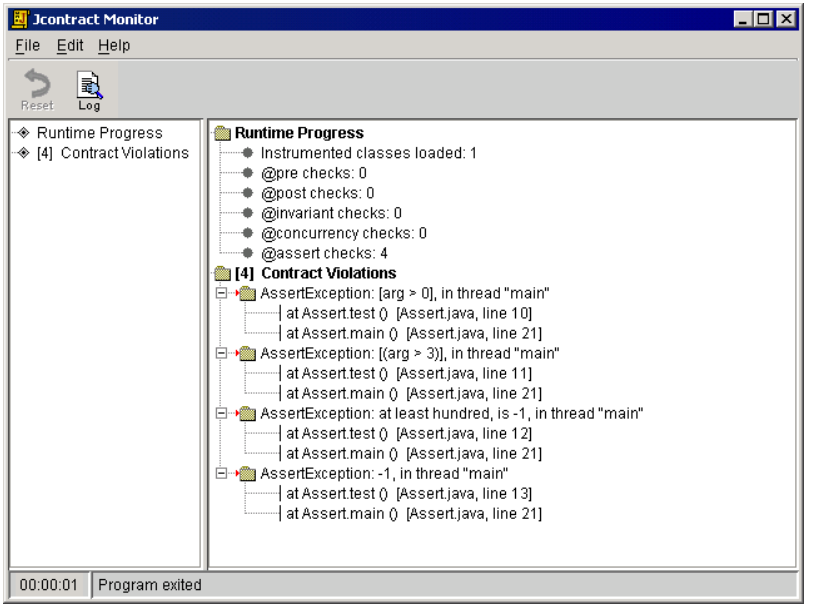
- **Runtime Progress:** Reports the number of instrumented classes loaded and the number of contracts executed.
- **Contract Violations:** Each time a contract violation occurs, the Jcontract GUI Monitor reports it in the **Contract Violations** area. If you expand the violation report, the monitor will display the stack trace where the violation occurred.

To view the violating file's source code, with the stack trace line highlighted, double-click the node which represents the stack trace line that you want highlighted.

To edit the source code, right-click any line of the stack trace information, then choose **Edit Source** from the shortcut menu. This opens the code in the default editor (WordPad).

Note: The source viewer looks for source files in the CLASS-PATH environment variable. To specify additional directory where to look for source, set the SOURCEPATH environment variable.

If you would like to hide either the Runtime Progress information or the Contract Violations information, clear the appropriate button on the left side of the GUI.


Jcontract Monitor
File Edit Help
Reset
Log
Runtime Progress
[4] Contract Violations
Runtime Progress
Instrumented classes loaded: 1
@pre checks: 0
@post checks: 0
@invariant checks: 0
@concurrency checks: 0
@assert checks: 4
[4] Contract Violations
AssertException: [arg > 0], in thread "main"
at Assert.test () [Assert.java, line 10]
at Assert.main () [Assert.java, line 21]
AssertException: [(arg > 3)], in thread "main"
at Assert.test () [Assert.java, line 11]
at Assert.main () [Assert.java, line 21]
AssertException: at least hundred, is -1, in thread "main"
at Assert.test () [Assert.java, line 12]
at Assert.main () [Assert.java, line 21]
AssertException: -1, in thread "main"
at Assert.test () [Assert.java, line 13]
at Assert.main () [Assert.java, line 21]
00:00:01
Program exited

The Menu Bar

The menu bar lets you access commands related to monitor functionality.

File Menu

- **Exit:** Closes that monitor GUI.

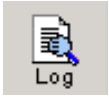
Edit Menu

- **Find:** Opens a dialog box that allows you to search for items in the monitor.
- **Preferences:** Opens a dialog box that allows you to modify Jcontract preferences. For information on available preference options, see "Jcontract Preferences" on page 37.

Help Menu

- **Contents:** Opens the Jcontract User's Guide.
- **Feedback:** Displays information about how to send feedback about Jcontract to ParaSoft.
- **Support:** Opens the Jcontract online support page.
- **About:** Displays the Jcontract version number and logo.

The Tool Bar

Button	Name	Action
	Reset	Resets all monitor counters and lists to 0.
	**Log**	Opens the Jcontract log file (this file is discussed in "The Log File" on page 21).

The Status Bar

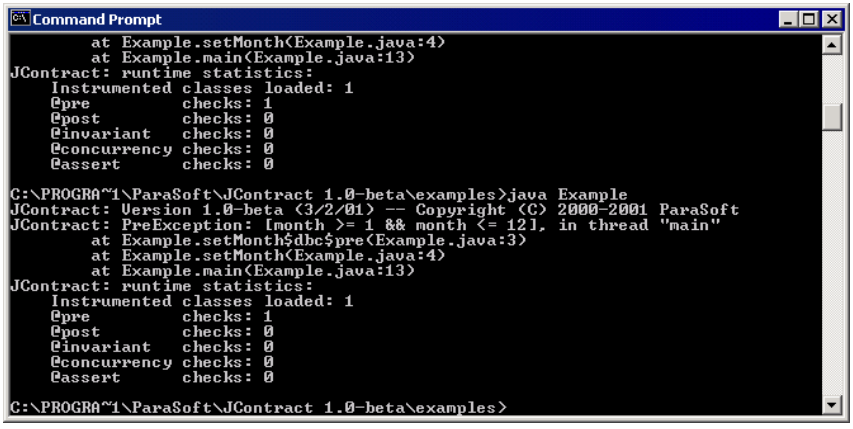
The status bar reports Jcontract messages.

TEXT Monitor

If the `Monitor.Type` preference in `<jcontract_install_dir>\u\<username>jcontract.preferences` is set to the value `Text`, the Jcontract TEXT Monitor will start as soon as the program under execution loads a class that contains instrumented contracts.

The TEXT Jcontract Monitor sends all messages to the console (stdout) by default. Use the `Monitor.LINEOutputFile` preferences in `<jcontract_install_dir>\u\<username>jcontract.preferences` to send the output to a file or to stderr.

When the instrumented program exits, the runtime progress at the point of exit is displayed (this feature requires running with JDK 1.3 or higher).



The Log File

All test parameters and results are recorded in the Jcontract log file.

You can access this log file by the clicking the **Log** button in the Jcontract GUI Monitor, or by opening the file directly (by default, it is named `jcontract.log` and is saved in the same directory as the program under test.

The log file contains the following sections:

- **Jcontract:** Lists Jcontract version information.
- **Environment:** Lists environment variables.
- **Started on**: Lists the time and date the test was started.
- **Jcontract installation directory:** Lists the Jcontract installation directory.
- **Jcontract Preferences:** Lists all preference option settings used for the current test.
- **Loaded instrumented class:** Lists the name of the class that Jcontract tested.
- **Jcontract Exceptions:** Lists details about any contract violations that occurred.
- **Jcontract Runtime Statistics:** Lists the number of classes loaded and the number of each type of check that was performed.
- **Ended on:** Lists the time and date the test ended.

jcontract.log - WordPad
File Edit View Insert Format Help
JContract: Version 1.0-beta -- Copyright (C) 2000-2001 ParaSoft
Environment:
java.version = 1.3.0
java.vendor = Sun Microsystems Inc.
java.home = C:\Program Files\JavaSoft\JRE\1.3
java.vm.version = 1.3.0-C
java.class.path = C:\Program Files\Exceed.nt\hcljrcsv.zip;C:\Program Files\Exceed.nt
java.ext.dirs = C:\Program Files\JavaSoft\JRE\1.3\lib\ext
os.name = Windows 2000
os.arch = x86
os.version = 5.0
user.name = cynthia
user.home = C:\Documents and Settings\cynthia
user.dir = C:\Program Files\ParaSoft\JContract 1.0-beta\examples
Started on: 3/5/01 11:24 AM
JContract installation directory:
C:\Program Files\ParaSoft\JContract 1.0-beta
JContract Preferences:
Instrumentation.DefaultConcurrency=CONCURRENT
Instrumentation.InstrumentAssertConditions=true
Instrumentation.InstrumentConcurrencyConditions=true
Instrumentation.InstrumentInvariantConditions=true
Instrumentation.InstrumentPostConditions=true
Instrumentation.InstrumentPreConditions=true
Instrumentation.InstrumentThrowsConditions=true
Instrumentation.InstrumentVerboseStatements=true
Instrumentation.LogFile=dbc_javac.log
Instrumentation.RuntimeHandler=jcontract.MonitorRuntimeHandler
Instrumentation.TmpDirectory=\Temp
Instrumentation.WriteLog=true
Monitor.GUI.Bounds=java.awt.Rectangle[x=363,y=341,width=...
Monitor.TEXT.Output=stdout
Monitor.TEXT.OutputPathFile=jcontract.out
Monitor.Type=GUI
Loaded instrumented class: Example
JContract: PreException: [month >= 1 && month <= 12], in thread "main"
at Example.setMonthdbcpre(Example.java:3)
at Example.setMonth(Example.java:4)
at Example.main(Example.java:13)
JContract: runtime statistics:
Instrumented classes loaded: 1
@pre checks: 1
@post checks: 0
@invariant checks: 0
@concurrency checks: 0
@assert checks: 0
Ended on: 3/5/01 11:24:35 AM
For Help, press F1

About Design by Contract

Design by Contract is a structured way of writing comments to define what code should do. The contract requires components of the code (such as classes or methods) to follow certain specifications as they interact with each other. The interactions between these components must fulfill a set of predetermined mutual obligations.

Design by Contract originated in Eiffel. Eiffel classes are components that cooperate through the use of the contract, which defines the obligations and benefits for each class. DbC is not yet commonly a part of programming languages such as C, C++, and Java, but ideally it should be. After all, any piece of code in any language has implicit contracts attached to it. The simplest example of an implicit contract is a method to which you are not supposed to pass `null`. If this contract is not met, a `NullPointerException` will occur. Another example is a component whose specification states that it only returns positive values. If it occasionally returns negative values and the consumer of this component is expecting the functionality described in the specification (only positive values returned), this contract violation could lead to a critical problem in the application.

Tools like Jtest and Jcontract bring Design by Contract to Java by helping you specify the contracts in comments and check whether or not the contract has been fulfilled.

Example

This is an example of a class with Design by Contract comments.

```
public class ShoppingCart
{
    /**
     * @pre item != null
     * @post $result > 0
     */

    public float add (Item item) {
        _items.addElement (item);
        _totalCost += item.getPrice ();
```

```
        return _totalCost;
    }
    private float _totalCost = 0;
    private Vector _items = new Vector ();
}
```

The contract specifies:

1. A precondition ("@pre item != null") which specifies that the item to be added to the shopping cart shouldn't be "null".
2. A postcondition ("@post $result > 0") which specifies that the value returned by the method should always be greater than 0.

Preconditions and postconditions can be thought of as sophisticated assertions. Preconditions are conditions that the client of the method needs to satisfy in order for the method to work properly. Postconditions are conditions that the implementor of the class guarantees will always be satisfied.

Benefits

Benefits of using DbC include:

- The code's assumptions are clearly documented (for example, you assume that `item` should not be `null`). Design concepts are placed directly in the code itself.
- The code's contracts can be checked for consistency because they are explicit.
- The code is much easier to reuse.
- The specification will never be lost.
- When you see the specification while writing the code, you are more likely to implement the specification correctly.
- When you see the specification while modifying code, you are much less likely to introduce errors.

Once you start using Jtest and Jcontract, the benefits of using DbC also include:

- Black-box test cases are created automatically. If you currently create your black-box test cases manually, this means fewer resources spent creating test cases and more resources you can dedicate to more complex tasks, such as design and coding. If you do not currently perform black-box testing, this will translate to more reliable software/components.
- Black-box test cases are automatically updated as the code's specification changes.
- Class/component misuse is automatically detected.
- The class implementation can assume that input arguments satisfy the preconditions, so the implementation can be simpler and more efficient.
- The class client is guaranteed that the results will satisfy the post-conditions.

For More Information

For more information about DbC see:

- http://www.eiffel.com/doc/manuals/technology/contract/page.html
- http://www.elj.com/eiffel/feature/dbc/java/ge/
- http://www.sdmagazine.com/articles/2000/0007/0007c/0007c.htm

Note: "Design by Contract" is a trademark of Interactive Software Engineering.

The Design by Contract Specification Language

This document describes the syntax and semantics for the Design by Contract (DbC) specification supported by Jtest and Jcontract.

The Design by Contract contracts are expressed with Java code embedded in Javadoc comments in the .java source file.

This document is divided into the following sections:

- “Tags Used for Design by Contract” on page 26
- “Contract Syntax” on page 31
- “Contract Inheritance” on page 34
- “Coding Conventions” on page 35

Tags Used for Design by Contract

The reserved Javadoc tags for DbC are:

- @invariant: Specifies class invariant condition.
- @pre: Specifies method precondition.
- @post: Specifies method postcondition.
- @concurrency: Specifies the method concurrency.

Other tags supported by Jtest and Jcontract include:

- @throws/@exception: Used to document exceptions.
- @assert: Used to add assertions in the method bodies.
- @verbose: Used to add verbose statements to the method bodies. (Not currently used by Jtest)

The following subsections describe each DbC tag in detail.

@pre

Description

Pre-conditions check that the client calls the method correctly.

Point of execution

Right before calling the method.

Scope

Can access anything accessible from the method scope except local variables. For example, it can access method arguments, and methods/fields of the class.

@post

Description

Post-conditions check whether the method works correctly.

Sometimes when a post-condition fails it means that the method was not actually supposed to accept the arguments that were passed to it. The fix in this case is to strengthen the precondition.

Point of execution

Right after the method returns successfully. Note that if the method throws an exception the @post contract is not executed.

Scope

Same as @pre, plus it can access "$result" and "$pre (type, expression)".

Accessibility

Same as @pre.

@invariant

Description

Class invariants are contracts that the objects of the class should always satisfy.

Point of execution

Same as @pre/@post: invariant checked before checking the precondition and after checking the postcondition.

Done for every non-static, non-private method entry and exit and for every non-private constructor exit.

If a constructor throws an exception, its @invariant contract is not executed.

Not done for "finalize ()".

When inner class methods are executed, the invariants of the outer classes are not checked.

Scope

Class scope, can access anything a method in the class can access, except local variables.

Accessibility

Same as @pre/@post.

@concurrency

Description

The @concurrency tag specifies how the method can be called by multiple threads. Its possible values are:

- **Concurrent:** The method can be called simultaneously by different threads (i.e., the method is multi-thread safe). Note that this is the default mode for Java methods.
- **Guarded:** The method can be called simultaneously by different threads, but only one will execute it in turn, while the other threads will wait for the executing one to finish. In other words, it specifies that the method is synchronized. Jcontract will only

report a compile-time error if a method is declared as “guarded” but is not declared as “synchronized”.

- **Sequential:** The method can only by executed by one thread at once and it is not declared synchronized. It is thus the responsibility of the callers to ensure that no simultaneous calls to that method occur. For methods with this concurrency contract, Jcontract will generate code to check if they are being executed by more than one thread at once. An error will be reported at runtime if the contract is violated.

Point of execution

Right before calling the method.

@throws/@exception

These are the standard @throws and @exception tags found in Javadoc; they are used to document that the method throws a given exception. @throws and @exception are synonymous. In this entry, we use @throws to represent both tags.

The syntax for the @throws tag is:

```
    ThrowsContract
        : @throws ExceptionName Text
```

Example:

```
/** @throws NegativeArraySizeException if size is negative */
```

When a method throws an exception, the Jcontract Runtime Handler will call 'documentedExceptionThrown (Throwable t)' if that exception is documented with a @throws tag.

Note that the Runtime Monitors provided with Jcontract don't take any action when 'documentedExceptionThrown' is called. You can nevertheless take a specific action by defining a user defined Runtime Handler.

Jtest suppresses exceptions that are documented with the @throws tag as long as the the classes were instrumented with the instrument @throws condition preference set to “true”.

@assert

Syntax

The syntax for the @assert tag is:

```
AssertStmt
        : @assert BooleanExpression
        | @assert '(' BooleanExpression ')'
        | @assert '(' BooleanExpression , MessageExpression ')'
```

The MessageExpression can be of any type.

For example:

```
/** @assert value > 0 */
/** @assert (value > 0) */
/** @assert (value > 0, "value should be positive */
/** @assert (value > 0, value) */
```

The @assert tags should appear in Javadoc comments inside the method bodies. If the classes are compiled with 'dbc_javac' and the Instrument.InstrumentAssertConditions preference is true/enabled, then the @assert boolean expression will be evaluated. If the expression evaluates to false, then one or more of the following actions take place:

- An error message is reported in Jtest's **Design by Contract> @assert** Results panel/Errors Found panel branch or in the Jcontract Monitor.
- A runtime exception (jcontract.AssertException) is thrown.
- The program exits by invoking System.exit (1).

See “Contract Semantics” on page 33 for more information about how to select the actions that take place. The default action is to report an error and continue program execution.

@verbose

The syntax for the @verbose tag is:

```
VerboseStmt
        : @verbose MessageExpression
```

```
        | @verbose '(' MessageExpression ')'
```

For example:

```
    /** @verbose "process starts" */
    /** @verbose ("process ends") */
    /** @verbose 26.7 */
```

The @verbose tags should appear in Javadoc comments inside the method bodies. If the classes are compiled with 'dbc_javac' and the Instrument.InstrumentVerboseConditions preferences is true/enabled, then the classes are instrumented with the verbose expression.

By default, all verbose statements are inactive; once they are activated, they print the MessageExpression to System.out.

The @verbose statements can be separately activated for each class.The @verbose statements for a class are active if the system property jcontract.verbose.CLASSNAME is set to the value ON (where CLASSNAME is the name of the class without the package part). For example, to activate the verbose statements in class pkg.DataDictionary on Windows use:

```
$ java -Djcontract.verbose.DataDictionary=ON ...
```

Note that the MessageExpression in a verbose statement is not evaluated if the verbose statement is inactive.

Contract Syntax

The general syntax for a contract is:

```
        DbcContract:
              DbcTag DbcCode
            | @concurrency { concurrent | guarded | sequential }
```

where

```
        DbcTag:
              @invariant
            | @pre
            | @post
```

```
DbcCode:
      BooleanExpression
    | '(' BooleanExpression ')'
    | '(' BooleanExpression ',' MessageExpression ')'
    | CodeBlock
    | $none

MessageExpression:
      Expression
```


Any Java code can be used in the DbcCode with the following restriction: the code should not have side effects (i.e., it should not have assignments or invocation of methods with side-effects).

The following extensions to Java (DbC keywords) are allowed in the contract code:

- $result: Used in a @post contract, evaluates to the return value of the method.
- $pre: Used in a @post contract to refer to the value of an expression at @pre-time. The syntax to use it is:
  `$pre (ExpressionType, Expression).`
  Note: The full "$pre (...)" expression should not extend over multiple lines.
- $assert: Can be used in DbcCode CodeBlocks to specify the contract conditions.
  The syntax to use it is:
  `$assert (BooleanExpression)`
  or
  `$assert (BooleanExpression , MessageExpression)`
- $none: Used to specify there is no contract.

Notes

- The @pre, @post and @concurrent tags apply to the method that follows in the source file.
- The MessageExpression is optional and will be used to identify the contract in the error messages or contract violation exceptions thrown. The MessageExpression can be of any type. If it is a

reference type it will be converted to a String using the "toString ()" method. If it is of primitive type it will first be wrapped into an object.

- There can be multiple conditions of the same kind for a given method. If there are multiple conditions, all conditions are checked. The conditions are ANDed together into one virtual condition. For example it is equivalent (and encouraged for clarity) to have multiple @pre conditions instead of a single big @pre condition.

Examples

```
    /**
      * @pre {
      *     for (int i = 0; i < array.length; i++)
      *         $assert (array [i] != null, "array elements
                   are non-null");
      * }
      */

    public void set (int[] array) {...}

    /** @post $result == ($pre (int, arg) + 1) */

    public int inc (arg) {...}

/** @invariant size () >= 0 */

    class Stack {...}

    /**
     * @concurrency sequential
     * @pre (value > 0, "value positive:" + value)
     */

    void update (int value) {...}
```

Contract Semantics

The contracts are specified in comments and will not have any effect if compiling or executing in a non DbC enhanced environment.

In a DbC-enhanced environment, the contracts are executed/checked when methods of a class with DbC contracts are invoked.

A contract fails if any of these conditions occur:

- The "BooleanExpression" evaluates to "false."
- An "$assert (BooleanExpression)" is called in a "CodeBlock" with an argument that evaluates to "false."
- The method is called in a way that violates its @concurrency contract.

If a contract fails, the Runtime Handler for the class is notified of the contract violation. Jcontract provides several Runtime Handlers; the default one uses a GUI Monitor that shows program progress and contract violations. You can also write your own Runtime Handlers; for details on how to do this, see “Runtime Handlers” on page 43.

With the Monitor Runtime Handlers provided by Jcontract, program execution continues as if nothing has happened when a contract is violated. For example, if a @pre contract is violated, the method will still be executed.

This option makes the DbC-enabled and non DbC-enabled versions of the program work in exactly the same way. The only difference is that in the DbC-enabled version, the contract violations are reported to the current Jcontract Monitor.

Note: Contract evaluation is not nested; when a contract calls another method, the contracts in the other method are not executed.

Contract Inheritance

Contracts are inherited. If the derived class or overriding method doesn't define a contract, it inherits that of the super class or interface. Note that a contract of $none implies that the super contract is applied.

If an overriding method does define a contract then it can only:

- Weaken the precondition: Because it should at least accept the same input as the parent, but it can also accept more.
- Strengthen the postcondition: Because it should at least do as much as the parent one, but it can also do more.

To enforce this:

- When checking the @pre condition, the precondition contract is assumed to succeed if any of the @pre conditions of the chain of overridden methods succeeds (i.e., the preconditions are ORed).
- When checking the @post condition, the postcondition contract is assumed to succeed if all the @post conditions of the chain of overridden methods succeed (i.e., the postconditions are ANDed).

Note: If there are multiple @pre conditions for a given method, the preconditions are ANDed together into one virtual @pre condition and then ORed with the virtual @pre conditions for the other methods in the chain of overridden methods.

For @invariant conditions, the same logic as for @post applies.

@concurrency contracts are also inherited. If the overriding method doesn't have an @concurrency contract, it inherits that of the parent. If it has an inheritance contract, it can only weaken it (as it does for @pre conditions). For example, if the parent has a “sequential” @concurrency, the overriding method can have a “guarded” or “concurrent” @concurrency.

Coding Conventions

When using Design by Contract in Java, the following coding conventions are recommended:

- Place all the @invariant conditions in the class Javadoc comment with the Javadoc comment appearing immediately before the class definition.
- Javadoc comments with the @invariant tag should appear before the class definition.
- All public and protected methods should have a contract. All package-private and private methods should also have a contract.
- If a method has a DbC tag, it should have a complete contract. This means that if you have both a precondition and a postcondi-

tion, you should use "DbcTag $none" to specify that a method doesn't have any condition for that tag.

- No public class field should participate in an @invariant clause. Because any client can modify such a field arbitrarily, there is no way for the class to ensure any invariant on it.
- The code contracts should only access members visible from the interface. For example, the code in a method's @pre condition should only access members that are accessible from any client that could use the method. In other words, the contract of a public method should only use public members from the method's class.

Note: Jcontract does not currently enforce these conventions.

Jcontract Preferences

User preferences for the Jcontract package are stored in the preferences file located in `<jcontract_install_dir>\u\<username>jcontract.preferences`.

That file contains preferences of the form:

```
Category.Item=value
```

For example:

```
Instrumentation.InstrumentPreConditions=true
```

Changing Preference Options

The preference values can be changed in either of the following ways:

- Edit the `jcontract.preferences` file directly.
- Edit the preference options through the Preference panel GUI interface that modifies the `jcontract.preferences` file. The are three ways to reach this Preferences panel:
    - Choose **Start> Programs> Jcontract> Edit Preferences**.
    - Enter the following command at the command prompt: `dbc_preferences`
    - (If the Jcontract Monitor is already open) Choose **Edit> Preferences** in the Jcontract Monitor.
- Edit the preference options with commands that work with the `dbc_preferences` command.
  For example, to change the `Instrument.InstrumentPreConditions` preference to `false` use the following command:
  `$ dbc_preferences -set Instrument.InstrumentPreConditions=false`
  For more information see “dbc_preferences” on page 50.

Available Preference Options

All of the following preference options can be modified in the Jcontract Preferences panel; all options that are not exclusive to the Jcontract GUI Monitor can be modified directly in the `jcontract.preferences file` or from the command line. The GUI options related to each preference option are listed in parentheses below the name of each option.

Instrumentation Preferences

Instrumentation.InstrumentPreConditions
(Instrumentation tab> Instrument @pre Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@pre" contracts in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@pre (on|off)" flag.

Instrumentation.InstrumentPostConditions
(Instrumentation tab> Instrument @post Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@post" contracts in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@post (on|off)" flag.

Instrumentation.InstrumentInvariantConditions
(Instrumentation tab> Instrument @invariant Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@invariant" contracts in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@invariant (on|off)" flag.

Instrumentation.InstrumentConcurrencyConditions
(Instrumentation tab> Instrument @concurrency Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@concurrency" contracts in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@concurrency (on|off)" flag.

Instrumentation.DefaultConcurrency
(Instrumentation tab> Default Concurrency)

Default concurrency to be used by "dbc_javac" for methods without a "@concurrency" contract.

Possible values are "concurrent", "guarded" and "sequential".The default value is "concurrent".

See “The Design by Contract Specification Language” on page 26 for more information about these values.

Note: This preference is not used for methods with empty bodies and get/set methods. Those methods are always assumed to have "concurrent" @concurrency if they don't have an explicit @concurrency tag.

Instrumentation.InstrumentThrowsConditions
(Instrumentation tab> Instrument @throws Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@throws" conditions in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@assert (on|off)" flag.

Instrumentation.InstrumentAssertConditions
(Instrumentation tab> Instrument @assert Conditions)

Specifies if the "dbc_javac" compiler should generate code to check the "@assert" conditions in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@assert (on|off)" flag.

Instrumentation.InstrumentVerboseStatements
(Instrumentation tab> Instrument @verbose Conditions)

Specifies if the "dbc_javac" compiler should generate code for the "@verbose" statements in the class.

Possible values are "true" and "false". The default value is "true".

The value specified here can be overridden in the "dbc_javac" command by using the "-Z@verbose (on|off)" flag.

Instrumentation.WriteLog
(Instrumentation tab> Write Log File)

Specifies if a log file should be written by the Jcontract runtime.

Possible values are "true" and "false". The default value is “true".

The value specified here is ignored if the “dbc_javac” flag "-Zruntime_preference Instrumentation.WriteLog..." was used with “dbc_javac” to compile the first instrumented class loaded.

Instrumentation.LogFile
(Instrumentation tab> Log File)

Specifies the path to the log file Jcontract writes.

The default value is "jcontract.log".

The value specified here is ignored if the flag "-Zruntime_preference Runtime.LogFile..." was used with “dbc_javac” to compile the first instrumented class loaded.

Instrumentation.TmpDirectory
(Instrumentation tab> Temporary Directory)

Specifies the directory where Jcontract writes the temporary files generated by "dbc_javac".

The default value is "\Temp".

Instrumentation.RuntimeHandler
(Instrumentation tab> Temporary Directory)

Specifies which Runtime Handler will be associated with the classes compiled with "dbc_javac".

The default value is "jcontract.MonitorRuntimeHandler".

The value specified here can be overridden in the "dbc_javac" command using "-Zruntime_handler {class_name}".

For more information about Runtime Handlers see “Runtime Handlers” on page 43.

Monitor Preferences

Monitor.Type
(Monitor tab> Monitor Type)

Specifies the type of monitor Jcontract uses at runtime.

The possible values are "TEXT" or “GUI". The default is "GUI".

See “Jcontract’s Monitors” on page 17 for more information about the Jcontract monitors.

Monitor.GUI.Bounds:
(Monitor tab> GUI Monitor Bounds)

Specifies the bounds of the Jcontract GUI Monitor.

This preference is updated automatically if the GUI monitor is moved or resized.

Monitor.TEXT.Output
(Monitor tab> TEXT Monitor Output)

Specifies the path to the file that the TEXT monitor will write to.

The possible values are "stdout", "stderr" and a file system path. The default value is "stdout".

For "stdout" and "stderr", the monitor sends the output to the standard output and standard error (the console).

Editor Preferences (GUI Only)

Editor
(Editor tab> Editor command)

Lets you specify which editor you want to Jcontract to invoke when you choose to edit files from the Jcontract GUI Monitor.

Runtime Handlers

The classes instrumented with Jcontract use a Runtime Handler to perform appropriate actions when a contract is violated and to gather statistics on the running program.

A Runtime Handler is a class extending "jcontract.RuntimeHandler". A single Runtime Handler object is created when the first instrumented class is loaded; this same Runtime Handler object is shared by all the classes that have the same Runtime Handler class associated to them.

Jcontract provides several built-in Runtime Handlers. If you would prefer to use a customized Runtime Handler, you can create one by extending "jcontract.RuntimeHandler".

The Runtime Handler associated with a class can be specified either in the Jcontract preferences file or using the -Zruntime_handler flag with "dbc_javac".

Runtime Handlers provided with Jcontract

jcontract.MonitorRuntimeHandler

This is the default Runtime Handler used. This Runtime Handler is completely non-intrusive. The instrumented program behaves exactly the same as the non-instrumented program. When a contract violation occurs, the violation is logged into the monitor, but program execution continues as if nothing has happened.

For more information about the monitors provided with Jcontract see “Jcontract’s Monitors” on page 17.

jcontract.BasicRuntimeHandler

With this Runtime Handler, a jcontract.ContractException is thrown whenever a contract is violated.

The exceptions thrown are:

Type of Violation	Exception Thrown
@pre violated	jcontract.PreException: ...
@post violated	jcontract.PostException: ...
@invariant violated	jcontract.InvariantException: ...
@concurrency violated	jcontract.ConcurrencyException: ...
@assert violated	jcontract.AssertException: ...

jcontract.DefaultRuntimeHandler

This Runtime Handler extends the functionality of the jcontract.BasicRuntimeHandler with:

- A logging mechanism: a file "jcontract.log" is generated while the program runs with logging information. This logging information includes the instrumented classes loaded, contracts violated, environment and statistics about the running program.
- A start message and an end message showing the program statistics.

To run programs with the "jcontract.MonitorRuntimeHandler" Jcontract must be installed in the target machine.

To run programs with the "jcontract.BasicRuntimeHanlder" and "jcontract.DefaultRuntimeHandler" the file "jcontract.jar" must be in the classpath of the target machine.

Multiple Runtime Handlers

A single Runtime Handler object is created when the first instrumented class is loaded and this same Runtime Handler object is shared by all the classes that have the same Runtime Handler class associated to them.

If there is more than one Runtime Handler class type in the running program a single Runtime Handler object is created for each type and those single instances are shared by all the classes that have the same runtime handler associated to them. The Runtime Handler that a class as associated is the one that was specified when the class was compiled using "dbc_javac".

Overriding the Runtime Handlers at Runtime

The Runtime Handler that was associated with a class when it was compiled with "dbc_javac" can be overrided by setting the property "jcontract.runtime.handler". For example if a program is run using:

```
$ java -Djcontract.runtime.handler jcontract.Default-
RuntimeHandler Main
```

then all the classes will use "jcontract.DefaultRuntimeHandler" as their Runtime Handler.

Generating Instrumented Classes that Require no Runtime

Jcontract can generate classes that require no runtime at all (i.e., classes that can be run without requiring "jcontract.jar" on the classpath).

To generate classes that require no runtime at all, use "-Zruntime_handler NONE" when compiling with "dbc_javac". Note that for those classes, the Runtime Handler cannot be specified by setting the property "jcontract.runtime.handler". Also, those classes will throw the following exceptions when a contract is violated:

Type of Violation	Exception Thrown
@pre violated	java.lang.RuntimeException: @pre: ...
@post violated	java.lang.RuntimeException: @post: ...
@invariant violated	java.lang.RuntimeException: @invariant: ...
@concurrency violated	java.lang.RuntimeException: @concurrency: ...
@assert violated	java.lang.RuntimeException: @assert: ...

User Defined Runtime Handlers

You can define your own Runtime Handler by writing a class that extends "jcontract.RuntimeHandler" and overriding at least the "contractViolation (RuntimeException ex)" method.

For an example see <jcontract_install_dir>\examples\UserDefinedRuntimeHandler.java.

For Jcontract API documentation, see <jcontract_install_dir>\docs\api\index.html.

dbc_javac

Name

dbc_javac - javac with “Design by Contract” enabled.

Synopsis

```
dbc_javac [dbc_javac options] [javac options]
```

Description

The dbc_javac command is a “Design by Contract” enabled replacement for javac. dbc_javac should be used in place of the javac command whenever you want the code contracts to be checked at runtime.

The dbc_javac command checks the DbC specification in the javadoc comments, generates instrumented .java files with extra code to check the contracts in the Javadoc comments and compiles the instrumented .java files with the javac compiler.

This process produces .class files instrumented with extra bytecodes to check the contracts at runtime. Unless `-Zruntime_handler NONE` is specified, `jcontract.zip` must be accessible at run time to the classes compiled with dbc_javac.

Options

The dbc_javac command accepts the same options as the javac command. In addition, it accesses some additional options of the form “-Z...”:

`-Z@(pre|post|invariant|concurrency|throws|assert|verbose) (on|off)`

> Controls whether the specific contract type is instrumented.
>
> Overrides the `Instrumentation.InstrumentXXXCondition` preference in the preferences file.

Example: `-Z@pre off`, if compiling with this option the resulting .class files will not check the @pre contracts.

`-Zdefault_concurrency (concurrent|guarded|sequential)`

Sets the default concurrency contract for methods without an explicit "@concurrency" contract.

Overrides the `Instrumentation.DefaultConcurrency` preference in the preferences file. Note that the default value in that file is `concurrent`, the default mode for Java methods.

`-Zstatistics`

Shows instrumentation statistics.

`-Ztimings`

Shows timing information.

`-Zverbose`

Shows the steps being taken. Also turns on `-Zstatistics` and `-Ztimings`.

`-Zlog (on|off)`

Set logging mode to `on` or `off`. Default is `on`.

Overrides the `Instrumentation.WriteLog` preference in the preferences file.

`-Zlog_file (path|stdout|stderr)`

Specifies the file to write the login info to.

Example: `-Zlog_file stdout`, will write logging info to the command console.

Overrides the `Instrumentation.LogFile` preference in the preferences file.

The default value in that file is `jcontract.log`.

`-Zpreferences_file path`

Specifies the preferences file to use. If this option is not set the default preferences file in `<jcontract_install_dir>\u\<username>jcontract.preferences` is used.

`-Zruntime_handler class_name`

Specifies the Runtime Handler that will be associated to the compiled classes. Overrides the `Instrumentation.RuntimeHandler` preference in the preferences file. The default value in that file is `jcontract.MonitorRuntimeHandler`. For more information about Runtime Handlers see “Runtime Handlers” on page 43.

Example: `-Zruntime_handle UserDefinedRuntimeHandler`, the classes compiled with this options will use as Runtime Handler an instance of the class `UserDefinedRuntimeHandler`.

`-Zjava_compiler {compiler_command}`

Uses `{compiler_command}` to compile the original and instrumented classes.

Example: `-Zjava_compiler jikes`. The default value for this flag is `javac`. Note that only `javac` if fully supported. If you have problems using another compiler, contact jcontract@parasoft.com.

See Also

“dbc_preferences” on page 50

“Jcontract Preferences” on page 37

“Jcontract’s Monitors” on page 17

“Runtime Handlers” on page 43.

dbc_preferences

Name

dbc_preferences - command line editor for "jcontract.preferences"

Synopsis

dbc_preferences [options] [preferences_file]

Description

Command line interface to edit the contents of the "jcontract.preferences" file. See “Jcontract Preferences” on page 37 for a list of the available preferences and their default values.

If "preferences_file" is omitted, the default preferences file in `<jcontract_install_dir>\u\<username>jcontract.preferences`" will be used.

Options

If the command is invoked without options, the Jcontract Preferences GUI interface to the Jcontract preferences file will be started.

`-reset`

Resets all preference values to their default values.

`-show`

Prints the contents of the preferences file.

`-set preference=value`

Changes the preference value.

Example: $ dbc_preference -set Intrumentation.DefaultConcurrency=sequential

See Also

“Jcontract Preferences” on page 37

“dbc_javac” on page 47.

Index

Symbols

@assert 30
@concurrency 28
@exception 29
@invariant 27
@post 27
@pre 27
@throws 29
@verbose 30

A

assertion 30

C

concurrency 9, 28
contacting ParaSoft 4
customizing Jcontract 37, 43

D

dbc_java command 47
dbc_preferences 50
Design by Contract
    coding conventions 35
    contract inheritance 34
    contract semantics 33
    contract syntax 31
    introduction 23
    specification 26
    tags 26

E

editor, specifying 42
environment, setting 6
exception 29

I

installation 3
instrumentation preferences 38
introduction 1
invariant 28

L

license 3
log file 21

M

man pages
    dbc_java 47
    dbc_preferences 50
monitors 17
    GUI 17
    preferences 41
    TEXT 20

P

ParaSoft, contacting 4
post-condition 27
pre-condition 27
preferences 37
    editor 42
    instrumentation 38
    monitor 41

Q

Quality Consulting 4

R

Running Jcontract
    example 10
runtime
    omitting 45
runtime handlers 43
    built-in 43
    customized 46
    multiple 44
    overriding 45

T

technical support 4
throws 29

U

Using Jcontract 6

V

verbose 30